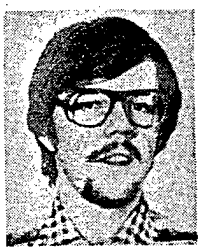
[16] B. Liskov, "Primitives for distributed computing," Distinguished Lecture Series, Carnegie-Mellon Univ., Pittsburgh, PA, 1980.
[17] P. H. Feiler, "IPC system version 1," Gandalf Internal Documentation, 1979.

Raul Medina-Mora (S'81) was born in Mexico City, Mexico in 1953. He received the B.S. degree in applied mathematics (Actuario) from the National University of Mexico, Mexico City, Mexico, and the M.S. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1976 and 1979, respectively.

Since 1975 he has been a Research Assistant in the Department of Computer Science, Institute for Applied Mathematics and Systems, National University of Mexico, and is currently on leave at Carnegie-Mellon University. He has been a graduate student at Carnegie-Mellon University since September 1976 and is currently finishing his Ph.D. His current research interests include software engineering, programming environments, and specifically, syntax-directed editing. He is presently working with the Gandalf Project in the design and implementation of a program development environment.

Mr. Medina-Mora is a student member of the Association for Computing Machinery.



Peter H. Feiler was born in Bad Toelz, Federal Republic of Germany, in 1952. He received the Vordiplom in mathematics and computer sciences from the Technical University in Munich in 1973.

Since September 1974 he has been a graduate student in computer sciences at Carnegie-Mellon University, Pittsburgh, PA, and is currently completing the Ph.D. degree. Since December 1980 he has been employed by Siemens Corporation with residence at Carnegie-Mellon University. He participated in the Family of Operating Systems project, in the design of STAROS, a multiprocessor operating system, and is currently involved in the design and implementation of a program development support environment (Gandalf Project). Other research interests include personal computing and local networks.

Mr. Feiler is a member of the Association for Computing Machinery.

An Experiment in Small-Scale Application Software Engineering

BARRY W. BOEHM

Abstract—This paper reports the results of an experiment in applying large-scale software engineering procedures to small software projects. Two USC student teams developed a small, interactive application software product to the same specification, one using Fortran and one using Pascal. Several hypotheses were tested, and extensive experimenal data collected. The major conclusions were as follows.

- Large-project software engineering procedures can be cost-effectively tailored to small projects.
- The choice of programming language is not the dominant factor in small application software product development.
- Programming is not the dominant activity in small software product development.
- The "deadline effect" holds on small software projects, and can be used to help manage software development.
- Most of the code in a small application software product is devoted to "housekeeping."

The paper presents the experimental data supporting these conclusions, and discusses their context and implications.

Index Terms—Programming languages, programming methodology, software engineering, software management, software project data.

Manuscript received April 18, 1980; revised December 29, 1980.

The author is with the Systems Engineering and Integration Division, TRW, Redondo Beach, CA 90278.

I. Introduction

Background

THE experiment described in this paper took place as part of a first-year graduate course in software engineering given at the University of Southern California (USC) in the Fall of 1978. It involved the development of a small (2000 deliverable source instructions) application software product: an interactive version of the COCOMO [1] model for estimating software costs. Two teams specified and developed independent versions of the same product, one team using Fortran and the other using Pascal.

The main reason for the project was to give the students experience in applying all the disciplines involved in practical software engineering: project planning, requirements specification, design, programming, testing, maintenance, management, technical communication, and human engineering of the man–machine interface. The choice of a cost estimation model as the product to be developed was based on three main criteria.

1) Its size appeared appropriate for the one-semester course schedule.

2) The subject matter was easy for students to understand.

0098-5589/81/0900-0482$00.75

3) The subject matter reinforced other material presented in the course.

The project also served as a useful vehicle for investigating the phenomenology of small application software development projects. To date, most software engineering data have been collected and analyzed on large projects [2]-[5] largely because the management of larger projects requires more data to be collected. To guide the investigation of the similarities and differences between large-scale and small-scale software phenomenology, several hypotheses were formulated and tested during the project.

1) Large-project software engineering procedures can be cost-effectively tailored to small projects.

2) The choice of programming language is the dominant factor in small software product development.

3) Programming is the dominant activity in small software product development.

4) The "deadline effect" holds on small software projects and can be used to help manage software development.

5) Most of the code in a small application software product is devoted to "housekeeping."

6) The COCOMO model provides an accurate estimate of the manhours required for small software product development.

Contents of Paper

Section II of this paper discusses the nature and significance of the hypotheses investigated during the project. Section III describes the project environment and experimental procedures. Section IV describes the project's progress and results by phase. Section V presents and discusses the results of testing each hypothesis. Section VI presents the conclusions resulting from the experiment.

II. Discussion of Hypotheses Investigated

Hypothesis 1: Large-project software engineering procedures can be cost-effectively tailored to small projects.

A number of software engineering techniques have been found to be highly cost-effective on large software projects, such as early project planning, thorough requirements and design specification and validation, early development of user's manuals, use of unit development folders and configuration management techniques, and independent product testing [6], [7]. In general, these techniques require a large amount of documentation and early nonprogramming activity on large projects. To date, there has been a wide divergence of opinion as to whether these techniques can be scaled down for small projects so that their benefits can be realized without overburdening the project with paperwork.

Hypothesis 2: The choice of programming language is the dominant factor in small application software product development.

The choice of programming language has a very strong effect on the decisions made during the programming portions of software development. On small projects, which devote more of their activity to programming than do large projects, the choice of language might be the dominant factor in determining the project's degree of success. On the other hand, there might be other factors such as program specification, management planning, or user engineering which have a larger influence on the project's outcome.

Hypothesis 3: Programming is the dominant activity in small software product development.

This hypothesis is somewhat related to the previous one, but deals more with the relative amount of effort devoted to programming during the software development process. Here "programming" is defined to include those development activites devoted to generating, compiling, debugging, and modifying code (including commentary) but excluding such activities as design, testing, walkthroughs, and noncommentary documentation. On large projects, programming activities typically consume about 7-10 percent of the total development effort [8]. On small projects, the percentage should be higher.

Hypothesis 4: The "deadline effect" holds on small software projects and can be used to help manage software development.

The "deadline effect" holds that the amount of energy and effort devoted to an activity is strongly accelerated as one approaches the deadline for completing the activity. It is clearly highly correlated with Parkinson's law: "Work expands to fill the available volume."

On a software project with a single-deadline at the completion of development, there is a high risk that even the accelerated effort of the development team will not be enough to meet the scheduled deadline. At this point, one has only two choices.

1) Slip the schedule.

2) Add more people in an attempt to meet the schedule, resulting in an even larger schedule slip (by Brooks' law [9]: "Adding more people to a late software project makes it even later").

However, if one can define a series of intermediate deadlines, or milestones, which contribute directly to the success of the overall project, one can use the deadline effect to generate enough intermediate effort to keep the project on schedule.

Where does this extra effort come from? Some of it may come from people spending extra hours on evenings and weekends. However, a good deal of it generally comes from a redistribution of the normal slack activities in the software engineer's workday. A Bell Labs time-and-motion study of 70 programmers [10] indicated that about 30 percent of the workday was spent in slack-type activities (see Fig. 6 for details). The existence of a project deadline has the effect of deferring some of the slack activities until after the deadline.[1]

Hypothesis 5: Most of the code in a small application software product is devoted to "housekeeping."

In sizing and cost estimation of large projects at TRW, we have found that there has been a tendency to underestimate the amount of code devoted to "housekeeping": user amenities, error processing, mode management, moving data around.

[1]Managers should be warned not to try to eliminate this slack by imposing more and more deadlines. This route generally leads to decreased morale, decreased long-range technical skills, and increased personnel turnover.

The small projects developed in this experiment made it feasible to examine each line of source code and classify it with respect to its function, and thus to determine the size and nature of the "housekeeping" portions of the code.

Hypothesis 6: The COCOMO model provides an accurate estimate of the manhours required for small software product development.

The initial calibration of the COCOMO model was done with respect to a sample of mostly medium and large software projects. One of the final assignments for the student teams, used as a test of their maintenance activity, was to enter their own project descriptors into their version of the COCOMO model, to see how well its estimate correlated with the actual amount of effort they spent in development.

III. The Experimental Project

This section describes the experimental software project undertaken by the two teams in terms of the product to be developed, the development environment, the projects' organization and staffing, the project schedule, the software engineering techniques used, and the experimental data collection and analysis techniques used.

The Software Product

The COnstructive COst MOdel (COCOMO) developed by the two teams is a partly analytic, partly table-driven model. It accepts descriptions of software components in terms of their size and their ratings with respect to 16 cost driver-attributes (e.g., hardware constraints, database size, required fault-freedom, personnel experience, use of tools, and modern programming practices). It uses these to calculate the amount of effort (and resulting dollar cost) required to develop each component and the overall system, and provides a breakdown of the effort and cost into four major development phases.

The interactive version of the COCOMO model to be developed was to allow users to interactively specify their software product descriptions at the terminal, to provide appropriate error responses and "help" messages, and to produce the resulting cost estimates either at the terminal, on an off-line printer, or in a user file for later reference.

The Development Environment

The products were developed on the USC PDP-10 computer, using the TOPS-10 time-sharing operating system. The TOPS-10 software support was reasonably good in terms of text editing, code inventory management, and debugging aids. The computer system was not very reliable, however; its mean-time between failures averaged about 4–6 h during the programming and test phases. Further, it became harder and harder to obtain access to a terminal as usage from other courses built up during the semester. The Pascal group experienced some difficulties with the Pascal compiler, also.

Organization and Staffing

Twelve students signed up for the course. It was decided to organize them into two 6-person teams, a Project Manager and five Assistant Project Managers (APM's) responsible for:

- developing the cost model subsystem of the product;
- developing the user input/output subsystem;
- developing the user's file subsystem;
- verifying and validating (V + V) plans, specifications, and products;
- performing project planning and control (P + C) functions.

In addition, the Project Manager was to serve in a chief programmer capacity, in developing the top-level requirements, design, and code for the product. Also, the nonprogramming team members were required to perform a successful modification of the product after development was complete.

Table I shows the major responsibilities of each team member during the various phases of the project. The acronyms will be explained in the section below on "Software Engineering Techniques Used."

At the start of the semester, the students were given a short diagnostic test to determine their familiarity with various aspects of software engineering. They were also given a copy of Table I and asked to rank their preference for the various jobs. Team assignments were made with respect to three main criteria:

1) Student Preference: Each student's job was either his first or second choice.

2) Language Experience: Team P (the Pascal team) did not contain any students with no Pascal experience; Team F (the Fortran team) did not contain any students with no Fortran experience.

3) Experimental Balance: For example, Team P members had an average of 34 months of programming experience; Team F, 33 months.

Software Engineering Techniques Used

Early Requirements Specification and Validation: The first phase of the project was devoted to developing requirements specifications and draft user protocols; independently validating the requirements; and holding a plans and requirements review (PRR) to determine how to resolve problems found in validation.

Early Product Design and Verification: The second phase was devoted to developing and independently verifying a product design and a draft user's manual and holding a product design review (PDR) to resolve problems identified.

Structured Development Techniques: These included top-down design and development, program design language, structured code, program library functions, structured walkthroughs, and a democratic variant of a chief programmer team.

Unit Development Folders (UDF's) [11]: These are formalized programmer's notebooks with sections for the requirements, design, test plans, code, test results, and as-built documentation. In this project, separate UDF's were established for each subsystem. A cover sheet on each UDF is used to track progress with respect to the development schedule for the unit.

Independent Verification and Validation (V + V): One team member performed independent V + V of the requirements, design, and software products developed by the other team members. The code was to be furnished to the APM-V + V only after it had been demonstrated to satisfy a set of con-

TABLE I
PROJECT RESPONSIBILITIES BY PHASE

	OCT. 18 PRR	NOV. 8 PDR	DEC. 1 TEST BASELINE	DEC. 13 ACCEPT. TEST	
PROJECT MANAGER	• PROJECT PLAN • TOP-LEVEL RQTS. SPECS. • DRAFT USER PROTOCOLS	• MANAGE W.R.T. PLAN • SYSTEM-LEVEL DESIGN	• MANAGE W.R.T. PLAN • PROGRAM TOP-LEVEL SOFTWARE • INTEGRATE SUB-SYSTEMS	• MANAGE W.R.T. PLAN • CLOSE OUT ALL ACCEPTANCE ITEMS	
APM-P&C	• PROJECT PLAN • REPORTING SYSTEM • PUBLICATIONS	• MONITOR PLANNED PROGRESS • UDF, LIBRARY PROCEDURES • PUBLICATIONS	• MONITOR UDF'S, PROGRESS • RUN LIBRARY • SOFTWARE PROBLEM REPORT (SPR) SYSTEM	• RUN LIBRARY, SPR SYSTEM • PUBLICATIONS	• MODIFY SOFTWARE
APM-FILE	• FILE SUBSYS. RQTS. • DRAFT USER PROTOCOLS	• FILE SUBSYS. DESIGN, TEST PLANS, PORTIONS OF USER MANUAL	• PROGRAM FILE SUBSYSTEMS • SUPPORT INTEGRATION	• FIX SPR'S • AS-BUILT DOC'N.	
APM-I/O	• I/O SUBSYS. RQTS. • DRAFT USER PROTOCOLS	• I/O SUBSYS. DESIGN, TEST PLANS, PORTIONS OF USER MANUAL	• PROGRAM I/O SUBSYS. • SUPPORT INTEGRATION	• FIX SPR'S • AS-BUILT DOC'N.	
APM-MODEL	• MODEL SUBSYS. RQTS.	• MODEL SUBSYS. DESIGN, TEST PLANS, PORTIONS OF USER MANUAL	• PROGRAM MODEL SUBSYS. • SUPPORT INTEGRATION	• FIX SPR'S • AS-BUILT DOC'N.	
APM-V+V	• V+V RQTS. • PRELIM. ACCEPTANCE TEST PLAN	• V+V DESIGN • INTEG. + TEST PLANS • ACCEPT. TEST PLAN	• FUNCTIONAL CAPABILITY LIST PREP. • INTERNAL ACCEPTANCE	• RUN SYSTEM TESTS • ISSUE SPR'S	• MODIFY SOFTWARE

ditions derived from the requirements and design by the APM-V + V (a functional capability list).

Baseline Configuration Management (CM): After each review and after turnover of the code to the APM-V + V for independent testing, the master version of each item was to be turned over to the APM-P + C as the formal baseline version of the item. Thereafter, the baselined master version could be changed only by the APM-P + C, using the problem report system to record the reason for and nature of the change.

Experimental Data Collection Procedures

Hypotheses about the amount and distribution of effort were tested by having each team member fill out weekly time sheets indicating how many hours in each day were spent in performing various basic activities: reading, designing, planning, programming, documentation, testing, reviewing, meeting, and fixing. These were collected and analyzed by the APM-Planning and Control.

Hypotheses about error rates during the development process were tested by collecting and analyzing the problem report forms employed as part of the V + V process.

Hypotheses about the distribution of the code by function were tested by reading each segment of code and categorizing it by function: model calculations, getting user inputs, furnishing user outputs, control or mode management, user help-message processing, error processing, moving data around, formats and data declarations, and comments.

Hypotheses about the acceptability of the resulting products were tested by conducting an independent acceptance test, in which two members of the USC Computer Science faculty were given the product user's manual a day in advance, and then spent a 2 h session using the product the following day. A log of resulting problems and comments was made by each team during their session.

Hypotheses about the relative importance and efficacy of languages and other software engineering techniques were tested in a nondirected fashion. At the end of the course, each student wrote a ten-page project critique, addressing the question "if we were to do the project over again, how could we do it better?" The results of these critiques were analyzed for the degree of consensus among team members and between teams of the most important factors influencing the project results.

IV. Project Progress and Results by Phase

Plans and Requirements Phase

Both teams produced their requirements specifications and life-cycle master plans on time and in the formats provided. The small size of the product meant that the spec was about at the same level of detail usually given in a product design spec for a large product: variable names assigned, detailed data structures provided. Team P had carried their spec much further than Team F, which had not developed much detail in their intrateam interface specs. Team P's spec contained 24 pages and generated 41 problem reports; Team F's spec contained 18 pages and generated 31 problem reports. Both averaged 1.7 problem reports/page,[2] which is typical of the rate for large software systems [12].

[2]The problem reports gave each student an appreciation of the number of inconsistencies that must be resolved even in small group-produced specs, and of the value of resolving them early. For example, one variable (the number of subsystems to be costed by the user) was defined in six different ways throughout the spec: NSUB, NUSB, NUMSUB, N, NSUM, and TBD (to be determined).

Neither team produced very thorough draft user protocols, which would turn out to cause a number of problems later.

During this phase, one Team P member dropped the course. This problem was handled by the instructor providing Team P with the cost model requirements spec he was to have produced, by having the APM-file also design and code the cost model subsystem, and by somewhat reducing the functional capability required of Team P's file subsystem. Fortunately, this did not seriously unbalance the experiment, and no further dropouts occurred.

Product Design Phase

Both teams produced their product design specifications and draft user's manuals on time and in the formats specified. Both teams required additional time for their draft test plans, mainly because the V + V persons did not get the product design spec early enough. Again, the small size of the product meant that the spec was at the level of detail usually given in a detailed design spec for a large product, e.g., detailed PDL for each routine.

In this phase, Team F had to "play catchup" both to refine their requirements spec and to reflect the additional detail in their design. Largely as a result, their design error rate was higher: 71 problem reports in a 58-page spec (1.2 problems/page), compared to 48 problem reports in a 68-page spec (0.7 problems/page) for Team P.[3]

The draft user's manuals were lacking considerably in detail. Team P's draft was 9 pages, compared to 28 pages in their final user's manual. Team F's draft was 6 pages, compared to a final version of 20 pages. This lack of detail was to cause a number of problems which showed up in the user acceptance test.

Two highly critical design problems were detected in the design V + V activity. Both teams specified a highly inflexible lock-step method for the user to furnish inputs to the model (easy to program via DO, FOR, or WHILE loops, but hard on the user who wants to go back and revise an earlier input, and finds he cannot do it until he completes the entire input sequence). Also, Team F designed a highly inefficient storage/retrieval scheme which required frequent disk accesses for quantities which could have been kept in core. The teams' response to these problems was not to modify their designs but to wait and see how they worked out in practice. In the time available to complete the project, these decisions turned out to be irrevocable.

Programming Phase

Progress in this phase was slowed down by the unreliability of the PDP-10 system and the increasing unavailability of terminals. One structured walkthrough per team was performed in class. Team F's walkthrough found 12 (genuine, nontrivial, distinct) problems in a 77-line routine. Team P's walkthrough found 13 problems in an 87-line routine. Both of these work out to about one problem detected per $6\frac{1}{2}$ lines of code.

Based on later feedback from the routine originators, only three additional problems were found in these two routines later. This meant that the walkthroughs were 25/28 = 89 percent effective in detecting errors in these routines. These results differ considerably from those reported in a controlled experiment by Myers [13]: one problem detected in walkthroughs per 11 lines of code, with a 38 percent effectiveness in detecting errors. The difference is most likely explained by the fact that Myers' sample routine contained a number of considerably more subtle errors than are found in most applications software (e.g., 6 of the 15 errors in Myers' sample had gone undetected when a version of the program was generated using correctness-proof techniques [14], [15]).[4]

Neither team used walkthroughs extensively, however, due partly to the difficulties of scheduling them with students having different on-campus schedules. Also, the unit development folders were not as effective a means of development control, due largely to this reason.

Integration and Test Phase

Progress in this phase was slowed down even more because of the PDP-10 system's lack of reliability and terminal availability. Also, some quirks in the Pascal compiler created some integration difficulties for Team P. Both teams were only able to complete integration and test by getting everyone together for a crash team effort on the weekend before the acceptance test.

As a result, the test baseline was late and only partially established. This meant that the independent test activity was incomplete, that only a fraction of the problems thus discovered were fixed, and the problem report/configuration management system broke down. One result of this was that accurate statistics on the product test activity were not available.

Acceptance Test

Both teams provided the acceptance test user with a copy of the users' manual in advance, and were ready for the acceptance-test session on schedule. Both products performed acceptably under some thorough exercising by the professor-users, with only one abort due to a TOPS-10 system-level control option.

The majority of the problems noted by the users were deficiencies in the product's man–machine interface: unnecessary restrictions, confusing options, missing items, and unexplained actions. Some of these would have been easy to fix, but others would have required considerable effort to fix, such as the lock-step process through the input options and the slow performance of the disk-oriented data storage/retrieval routines in Team F's product.

Each user noted about 30 problems with the product he was exercising in his 2 h session. Table II summarizes the most significant problems with each product, with the most critical

[3]These are "genuine, nontrivial, distinct" problem reports, excluding reviewer misunderstandings, trivial typos, and duplicates. Problem reports were generated both by the APM-V + V and by the instructor, whose intent was to provide a uniform standard of review thoroughness.

[4]This shows that results on the relative efficacy of error-detection techniques are highly dependent on the sample of programs used to obtain the results, implying a need for considerable care in evaluating the representativeness of such results to one's own software context.

TABLE II
ACCEPTANCE TEST RESULTS

PROBLEM CATEGORY	TEAM P	TEAM F
UNNECESSARY RESTRICTIONS	• UPPER-CASE ONLY ON COMMANDS • LOCK-STEP PROGRESS THRU OPTIONS • VERBOSE PROMPTS	• UPPER-CASE ONLY ON COMMANDS • LOCK-STEP PROGRESS THRU OPTIONS • VERBOSE HELP MESSAGES • VERY SLOW PERFORMANCE • OVERLY REDUNDANT DATA ENTRY
CONFUSING OPTIONS	• TOP-LEVEL "HELP" REQUESTS • INCONSISTENT RATING DEFINITIONS	• USE OF UNEXPLAINED SYNONYMS (TABLES, PARAMETERS, COST DRIVERS) • BLANK ACCEPTED AS VALID NAME • AMBIGUOUS TERMINATION OPTION • POOR OUTPUT FORMATTING
MISSING ITEMS	• CONFIRMATION OF DELETIONS • USER MANUAL (UM) INDEX • UM-CARRIAGE RETURN CONVENTIONS • UM-EXPLAIN FILE USAGE • MISSING "HELP" OPTION	• UM-USER ID DEFINITION • MISSING "HELP" OPTIONS • UM-REAL/INTEGER CONVENTIONS • NO CHECK FOR MISSING INPUTS
UNEXPLAINED ACTIONS	• ASSIGNMENT OF DEFAULT NAMES • TRUNCATION OF ID'S	• NO "EXECUTION IN PROGRESS" MESSAGE
ERRORS	• RANGE CHECKING ON MODULE SIZING • UM-GIVEN RUN COMMAND WON'T WORK • "HELP" NOT TREATED AS COMMAND IN SOME MODES • ABORT IF TRY FILE IN WRONG FORMAT • NO WAY TO MODIFY MODULE SIZING	• RANGE-CHECKING ON MODULE SIZING • CONTROL-Z BLOW SYSTEM • ASSUMES RESPONSE IS "NO" IF NOT "YES"

problems underlined. "UM" in Table II stands for "Users' Manual," a source of several acceptance-test problems.

Maintenance

Each team was then given a "change order" to extend some of the cost calculations and provide some additional output cost estimate information and reports. The changes were performed by the nonprogramming members of the development team (V + V and P + C members), and were successfully completed on schedule. The extensive documentation and commentary along with the structured code were considered very helpful by the maintainers, although the incomplete configuration management caused some problems in using updated master versions of the various subsystems.

There was no significant difference in maintainability between the Fortran and Pascal versions. Both were well-structured; the Pascal version was better formatted, while the Fortran version had better commentary.

V. TEST OF HYPOTHESES

Hypothesis 1: Large-project software engineering procedures can be cost-effectively tailored to small projects.

The on-time, essentially successful completion and maintenance of both products with respect to an ambitious schedule and difficult host computer environment tends to confirm the hypothesis with respect to the effectiveness of the techniques.

However, the incomplete employment of the unit development folder, configuration management, structured walkthrough, and independent V + V techniques, largely due to time pressures, might lead one to conclude that these were considered not cost-effective by the two teams. Also, the large amount of time and effort devoted to producing requirements and design specifications might not have been cost-effective for a small product.

These conclusions were not substantiated by the project critiques produced by the participants. Table III presents an ordered list of the major improvement items listed in the critiques, in terms of the number of participants who included the item as a significant area for improvement in the project. The highest-ranking item was "more time for testing," generally with a comment that the independent testing activity should have had time to be fully exercised. However, the participants did not feel that too much effort had been put into the specification activities, as the next-ranking item was "more thorough specifications." The increased test time was rather suggested to come from "doing more work earlier" (item 6, cited by 5 participants) or "lengthening the development schedule" (item 14, cited by 3 participants). Only 2 participants felt that there was too much documentation involved (item 20).

Further, the other techniques whose use was reduced due to time pressures were considered by the participants as items which should have been improved. Near the top of the list were "full use of walkthroughs" (item 3, cited by 7 participants) and "full use of configuration management" (item 4, cited by 6 participants). The need for full use of unit development folders (item 11, cited by 3 participants) drew less of a consensus. This is probably due to two main reasons.

1) Their value in enhancing visibility is less on a small project.

2) Due to their novelty, they were not used to full advantage on the project.

One software engineering technique which generally works

TABLE III
IMPROVEMENT ITEMS MENTIONED IN PROJECT CRITIQUES

IMPROVEMENT ITEM	TEAM P (5 MEMBERS)	TEAM F (6 MEMBERS)	TOTAL
1. MORE TIME FOR TESTING	4	4	8
2. MORE THOROUGH SPECIFICATIONS	4	3	7
3. FULL USE OF WALKTHROUGHS	3	4	7
4. FULL USE OF CONFIGURATION MANAGEMENT	3	3	6
5. PDP-10 RELIABILITY, AVAILABILITY	3	2	5
6. DO MORE WORK EARLIER	3	2	5
7. MORE DELEGATION, COORDINATION BY MANAGERS	3	2	5
8. BETTER SPEC. OF USER INTERFACE	1	3	4
9. NEED FOR CLERICAL SUPPORT	3	0	3
10. COMPUTER SUPPORT OF LIBRARY FUNCTIONS	3	0	3
11. FULL USE OF UDF'S	2	1	3
12. STRENGTHEN MANAGER'S AUTHORITY	2	1	3
13. STRENGTHEN P+C, V+V ROLES	2	1	3
14. LENGTHEN DEVELOPMENT SCHEDULE	2	1	3
15. IMPROVE TEAM COMMUNICATIONS	1	2	3
16. IMPROVE TEAM-F DATA-ACCESS DESIGN	0	3	3
17. EQUALIZE ASSIGNMENTS	2	0	2
18. CLEAN UP PASCAL COMPILER PROBLEMS	2	0	2
19. MORE USER INTERACTION DURING PROJECT	1	1	2
20. REDUCE REQUIRED DOCUMENTATION	0	2	2
21. STRUCTURED CONTROL FEATURES IN LANGUAGE	0	1	1
22. MANY OTHER SINGLE COMMENTS	–	–	1

better on small projects than on large projects experienced some difficulties here. This was the chief programmer technique. Five participants, including both project manager/chief programmers, felt that the extensive specification and programming activities required of the chief programmer function detracted from the management coordination and communication activities needed in the project manager function, and that more of the leader's work should have been delegated. Both project leaders put in more time than anyone else on the team, averaging 17 h/week as compared to 10 h/week for the other team members. This sort of overload is generally the problem in applying the chief programmer technique on large projects, on which the team leaders have a heavy additional workload of interteam coordination as well [16].

On balance, the evidence tends to confirm Hypothesis 1: large-project software engineering techniques can be cost-effectively tailored to small projects. The main difficulty standing in the way of fully utilizing the techniques was the arbitrary schedule imposed by the USC semester. This precluded the teams from following the important axiom.

If there is a choice between "do it right" and "do it on schedule," by all means choose the former.

Hypothesis 2: The choice of programming language is the dominant factor in small software product development.

In all of the participants' critiques, the choice of programming language was mentioned only once (item 21 in Table III). The input-output subsystem programmer indicated that about 40-50 percent of the user input and error-checking logic consisted of Fortran GOTO's which could have been eliminated in a language containing CASE, IFTHENELSE, and BEGIN-END constructs, with an accompanying increase in code clarity.

Since this comment was strongly dominated in the critiques by participant comments on other software engineering and management items, and since the results of the acceptance test and maintenance activity were not strongly influenced by language features, it is evident that this experiment does not confirm Hypothesis 2. Rather, it is evident that many factors were more critical to project success than was the choice of programming language.

Hypothesis 3: Programming is the dominant activity in small software product development.

Fig. 1 shows the distribution of project effort[5] by activity for the two projects, based on the weekly timesheets filled out by the participants. The distributions for the two projects are strikingly similar. They indicate that programming consumes more of the total effort on small projects than on large projects (here, 17 percent and 12 percent versus 7–10 percent on large projects). But the programming effort is far from dominant. The largest effort by a good margin on both projects (32 percent and 28 percent) was devoted to documentation. One might consider that this might reflect an excessive level of documentation requirements, were it not that the participants' critiques summarized in Table III strongly favored more rather than less documentation (items 2 and 20).

Thus, it is evident that this experiment does not confirm Hypothesis 3. A number of other project activities require a comparable or greater amount of effort on a small project to that required for programming.

Hypothesis 4: The "deadline effect" holds on software projects and can be used to help manage software development.

Both projects had three major external deadlines; the plans

[5]The amounts shown for "fixing" are additional to the 100 percent accounted for by the other activities.

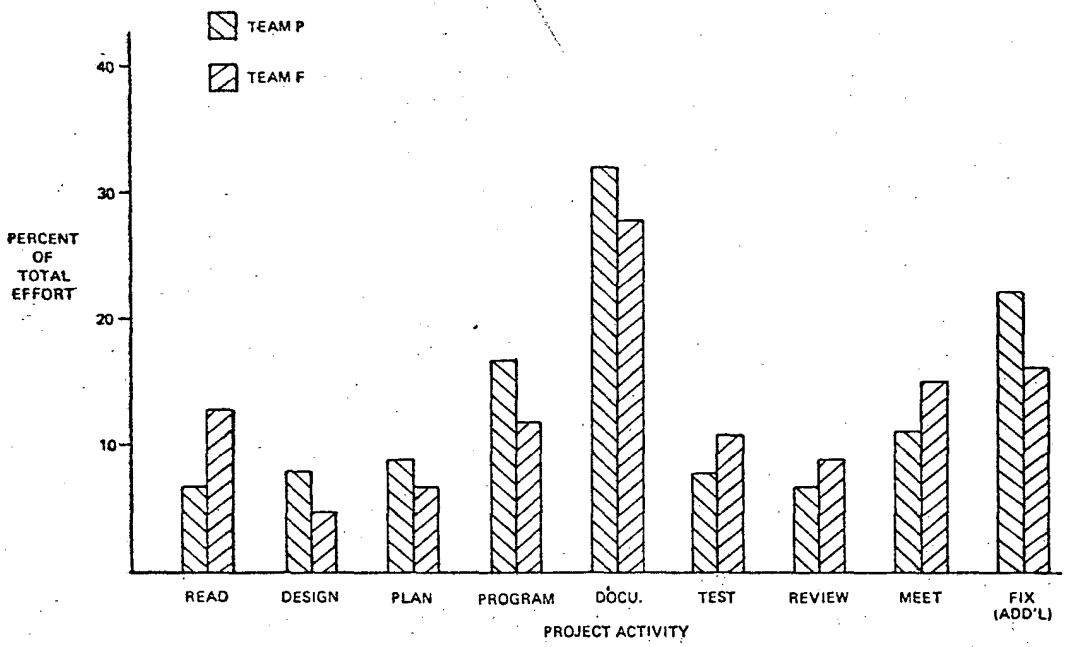


Fig. 1. Distribution of project effort by activity.

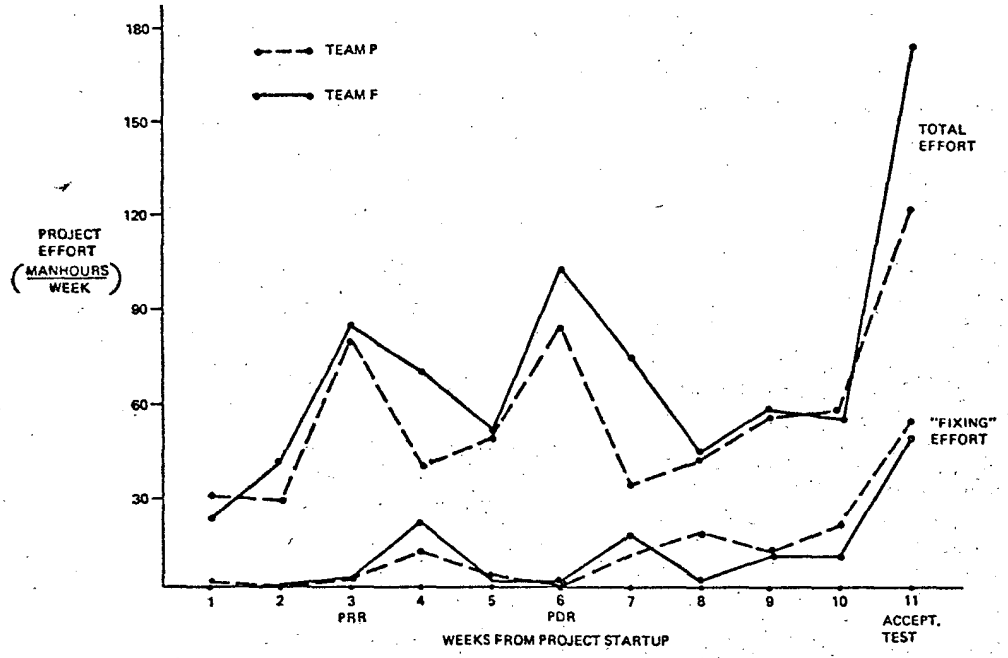


Fig. 2. Distribution of project effort by week.

and requirements review (PRR) at the end of Week 3, the product design review (PDR) at the end of Week 6, and the acceptance test at the end of Week 11. Fig. 2 shows the distribution of both project's effort by week, with the major deadlines indicated. It is clear that the "deadline effect" held for both projects, as the level of effort on both projects increases sharply in the weeks ended by the three major deadlines: Weeks 3, 6, and 11.

Thus, the intermediate deadlines of the PRR and PDR were used to help manage the project, by focusing more effort on early requirements and design problem identification and correction. This represents a significant savings in effort, since several organizations' experience on large projects has shown that these problems are much more expensive to correct (by factors of 10 to 100) in later phases [17]. On these projects, the cost-to-fix-increase between the requirements phase and the acceptance test phase was about 4:1 (see [1, p. 40]).

Fig. 2 also shows the number of hours per week devoted to *fixing* project plans, specifications, and code. Here again, the distributions for the two projects are roughly similar, with peaks in the weeks after PRR and PDR and in the final week of testing. Some differences are evident, due largely to the extra amount of fixing required by Team F to make up for the deficiences in its initial requirements spec. For completeness, the distribution of weekly effort by project activity is shown in Fig. 3 (for Team P).

Thus, evidence from this experiment tends to confirm Hypothesis 4 on the existence and utility of the "deadline effect."

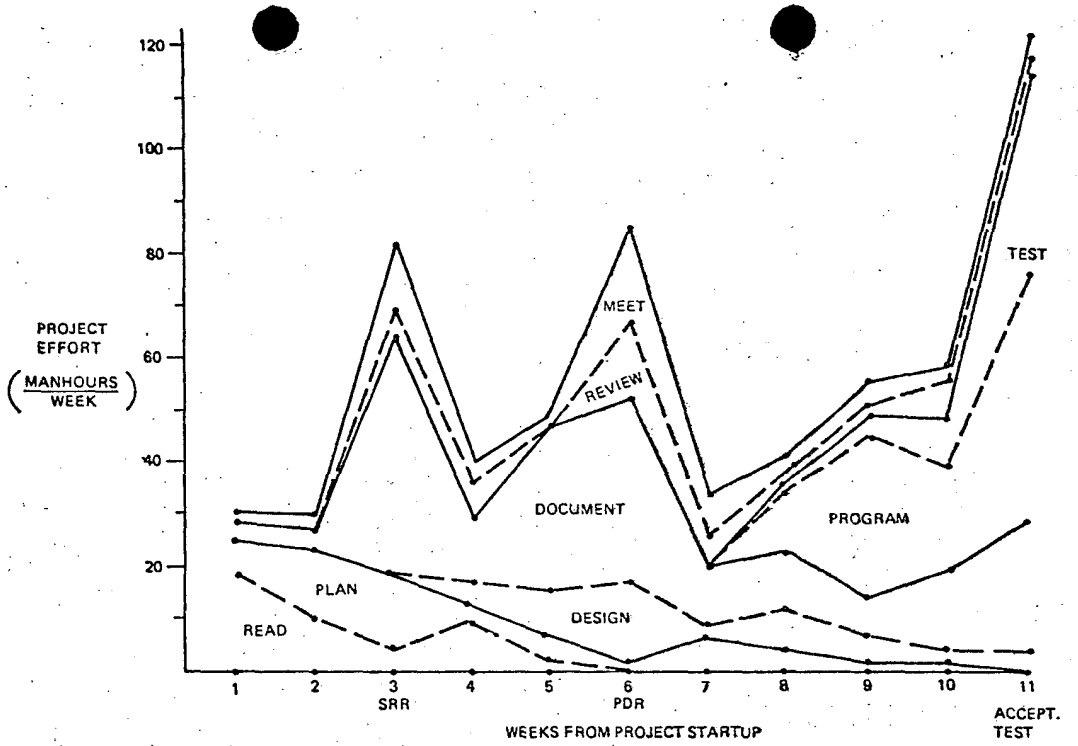


Fig. 3. Distribution of project effort by week and activity (team P).

On the other hand, the projects' effort curves clearly do not follow the Rayleigh distribution (Fig. 4) found to hold on some large projects [4], due to the multiple peaks caused by the deadlines.

Hypothesis 5: Most of the code in a software product is devoted to "housekeeping."

Fig. 5 shows the distribution of source code by function for the two projects.[6] Again, the distributions are markedly similar. The most striking observation is that the amount of code required to implement the actual cost model is only 2 percent for Team F and 3 percent for Team P. Far larger percentages are devoted to such "housekeeping" functions as error processing, mode management (control), user amenties, and moving data around.

The magnitude of this effect was a considerable surprise to me. For one thing, it showed how badly I had erred in distributing work assignments to the file, I/O, and model programmers. On both projects, the I/O subsystem programmer had about 50 percent of the lines of code to develop, while the model programmer had only about 10 percent (the other 40 percent were split in different ways on the two projects between the executive and file subsystems). This was one of the main reasons for the "equalize assignments" comments in the project critiques (Item 17 in Table III). Fig. 5 thus provides a useful start toward a general approach to project sizing, in giving an idea of the approximate percentages of a software product of this nature (a small, interactive computational model) which are devoted to certain classes of functions. Whether the distribution of source code by function is similar for other types of products is a useful subject for further investigation.

[6]The amounts shown for "comments" are additional to the 100 percent accounted for by the other functions.

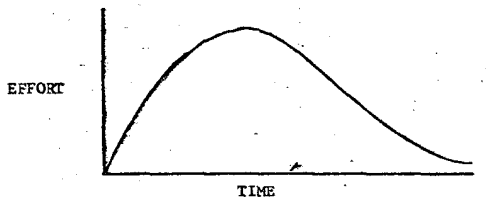


Fig. 4. A typical Rayleigh distribution.

Insights into "Software Piece-Part" Technology

Another useful subject for further investigation is the possibility of using the program function categories as a basis upon which to form libraries of standard program components or piece-parts. In analyzing and categorizing the source code of the two products, it appeared that much of the line-by-line code involved in obtaining user inputs, help message processing, report generation, and error processing could have been performed by a small number of standard parameterized procedures, and that much of the mode management (control) could have been handled by defining a hierarchy of modes and using decision tables to define mode transitions. The resulting code would have been considerably less efficient, but in an era of computational plenty and software personnel shortages, the reduction in lines of code to be programmed and tested would appear to make the effort worthwhile.

On the other hand, this sort of analysis is also useful in determining rough bounds on the potential payoff of a software piece-part technology. In analyzing the code involved in performing model calculations, moving data around, and defining data and formats, it appeared to be much more difficult to define standard program components which could lead to any significant reduction in the lines of code required to perform the functions. Since these latter functions accounted for

DISTRIBUTION OF SOURCE CODE BY FUNCTION

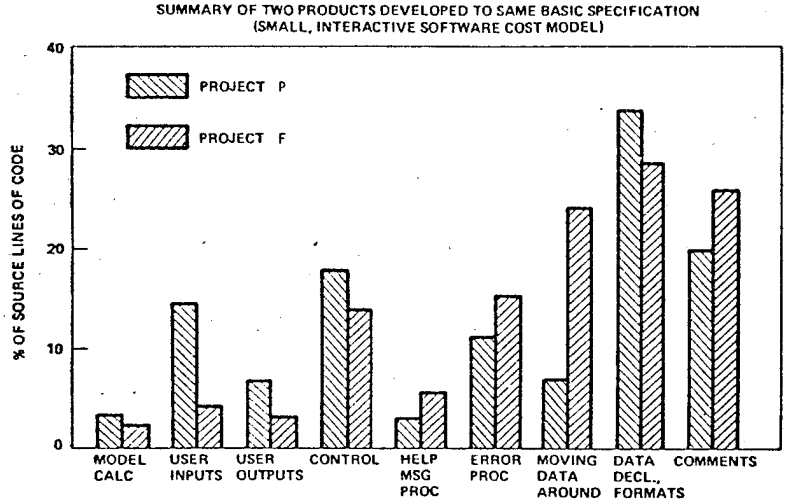


Fig. 5. What does a software product do?

about 50 percent of the total lines of code, it implies that the potential reduction in lines of code required to implement products of this nature is more like 50 percent than the 90 percent that one might hope for from software piece-part technology. Still a 50 percent potential reduction would be highly significant, indicating that further and more detailed investigations into the feasibility of software piece-part technology would be highly worthwhile.

Hypothesis 6: The COCOMO model provides an accurate estimate of the man-hours required for small software product development.

As a final test of their maintenance modifications, both teams entered a description of their own "interactive COCOMO" products into their version of the COCOMO model, to see how well the model would predict the magnitude of their efforts. The results are shown in Table IV.

At first glance, it would appear that the hypothesis was not confirmed: that the actual effort is only about 70 percent of that estimated by the COCOMO model. However, there may be a suitable explanation for this discrepancy. This is that the man-hours reported by the participants covered only activities directly related to the project. As seen in the summary of Bardain's study [10] in Fig. 6, these direct-project activities typically cover only about 70 percent of the total hours a programmer spends on the job. The other 30 percent of the time is consumed by "overhead" activities such as training, personal business, and nonproject communication.[7] Based on this rationale, one might be willing to say that Hypothesis 6 was provisionally confirmed, but it would be safer to say that the test was inconclusive.

A number of related statistics on the two projects are summarized in Table V. In general, there is a marked similarity between the sizes of the products and rates of production on the two products. Some other comparisons with related studies are as follows.

TABLE IV
COCOMO PREDICTIONS VERSUS ACTUAL EFFORT

Team	Man-Months		Actual / Predicted
	Actual	Predicted	
Team P	4.1	5.8	0.71
Team F	5.1	7.0	0.73

- The ratio of delivered source instructions to pages of documentation (DSI/page) of 15.5 is somewhat lower than the 20.4 DSI/page) reported by Felix and Walston [3] and considerably lower than the 25–48 DSI/page reported by Freburger and Basili [17]. Part of the difference is probably due to the larger sizes of their software products; part of it is due to differences in defining "documentation" [18].
- The number of documentation man-hours per page of documentation was 1.6, which works out to a cost of $40/page at a typical burdened rate of $25/man-hour. This is at the low end of the range of $35–150/page reported at an Air Force workshop [19]. Again, however, this $40 figure is not adjusted to include "overhead" time.

VI. CONCLUSIONS

The main conclusions are presented in terms of the hypotheses tested in the experiment.

Hypothesis 1: Large-project software engineering procedures can be cost-effectively tailored to small projects.

Result: Confirmed by this experiment, although some of the confirmation is based on participants indicating that they should have used some techniques more (independent test, walkthroughs, configuration management, unit development folders), rather than that they had fully used them and found them successful.

[7]Forgetting to account for this "overhead" factor is one of the four main reasons why people frequently underestimate software costs. The other three are underestimating the amount of code that will be required for "housekeeping" functions (Fig. 5); underestimating the amount of project effort devoted to project-oriented reading, planning, documenting, reviewing, meeting, and fixing (Fig. 1), and simply underestimating the number of application functions which the software product will actually require.

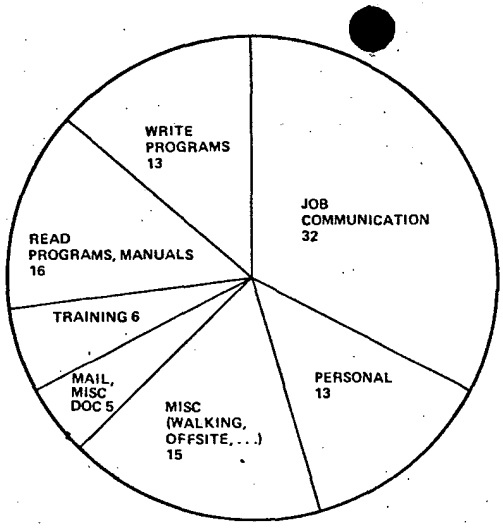


Fig. 6. What do programmers do? (Bell Labs time and motion study.)

TABLE V
COMPARATIVE PROJECT STATISTICS

ITEM	TEAM P	TEAM F
INITIAL REQUIREMENTS SPEC. – PAGES	24	18
PROBLEM REPORTS (PR'S/PAGE)	41 (1.7)	31 (1.7)
INITIAL LIFE CYCLE PLAN – PAGES	20	14
INITIAL RQTS./DESIGN SPEC. – PAGES	68	58
PROBLEM REPORTS (PR'S/PAGE)	48 (0.7)	71 (1.2)
INITIAL USER'S MANUAL – PAGES	9	6
FINAL LIFE CYCLE PLAN – PAGES	19	14
FINAL RQTS./DESIGN SPEC. – PAGES	73	79
FINAL USER'S MANUAL – PAGES	28	20
FINAL TEST PLAN – PAGES	11	6
FINAL TEST REPORT – PAGES	7	8
TOTAL FINAL DOCUMENTATION – PAGES	138	127
DOC'N. MANHOURS (MANHOURS/PAGE)	202 (1.5)	213 (1.7)
DELIVERED SOURCE INSTR. (DSI/PAGE)	2137 (15.5)	1977 (15.6)
TOTAL REPORTED MANMONTHS (DSI/MM)	4.1 (524)	5.1 (390)

Hypothesis 2: The choice of programming language is the dominant factor in small software product development.

Result: Not confirmed by this experiment. Many other software engineering and management factors were more critical to project success than was the choice of programming language.

Hypothesis 3: Programming is the dominant activity in small software product development.

Result: Not confirmed by this experiment. For the two teams, programming activities consumed 12 percent and 17 percent of the total development manhours, while documentation activities consumed 28 percent and 32 percent, and several other activities consumed about the same percentage of time as did programming.

Hypothesis 4: The "deadline effect" holds on small software projects and can be used to help manage software development.

Result: Confirmed by this experiment. Early deadlines stimulated efforts to validate requirements and design specs early, resulting in net project savings.

Hypothesis 5: Most of the code in a small application software product is devoted to "housekeeping."

Result: Confirmed by this experiment. For the two teams, the amount of code required to implement the actual cost model was only 2 percent and 3 percent of the total code developed. Far larger percentages were devoted to such "housekeeping" functions as error processing, mode management, user amenities, and moving data around.

Hypothesis 6: The COCOMO model provides an accurate estimate of the man-hours required for small software product development.

Result: Inconclusive. The data collected on the project did not include the "overhead" man-hours usually charged to projects. If adjustment is made for this overhead factor, there is a good agreement between the COCOMO estimates and the project results.

In addition, a number of figures and tables have been presented in the paper which provide some quantitative reference points for the distribution of effort, documentation, code, and problems during the phases of small application software development.

Some Qualifying Remarks

Choice of Programming Language: The results of this experiment do not imply that programming language issues are not important. The experimental results understate the importance of language issues in three main ways.

- The interactive cost model application involved largely simple programming constructs. Although most programs are just as straightforward, there are many more complex programs which are strongly benefited by modern language capabilities.
- The experiment involved a single stand-alone program. An organization with many programs to develop (and support) will find the choice of programming language a much more significant issue.
- There were no major applications-versus-language mismatches, as can occur in developing scientific programs in Cobol, for example.

The main result is thus not to downplay the importance of programming languages, but to emphasize that other factors are at least as critical to successful software engineering.

Productivity and Team Size: It is tempting to look at the comparative productivity of Team P (524 DSI/MM) and Team F (390 DSI/MM) in Table V, and ascribe the difference to the choice of programming language. However, the detailed activity statistics (and subjective impressions) on the projects indicate that the effect was due more to the additional person on Team F, since Team F's added man-hours were largely spent in group learning and coordination activities: reading, reviewing, and meeting.

Activity Man-Hours	Team P	Team F
Design, Plan, Program, Document, Test	467	477
Read, Review, Meet	144	294

The amount of time spent by both teams in direct product generation and test activities was about the same.

The amount of time spent on reading, reviewing, and meeting activities for both teams was a good deal higher than it would be for developing the same product in industry, for the following main reasons:

- for teaching purposes, the teams were larger than necessary;
- the students had no experience in working with each other, and little previous experience in working with groups.

"Deadline Effect": This effect was no doubt magnified by the student environment, which does not have the stabilizing factor of a 40 hour work week. However, it is impressive that the deadline effect held for the students even in the face of other patterns of demand on student time: deadlines in other courses, big-game weekends, qualifying exams, etc.

Preconditioning: The procedures established for the projects committed both teams to a number of activities (e.g., early reviews, baseline configuration management) and documentation requirements (e.g., life-cycle plans, requirements specifications, test plan) which are frequently not encountered on small projects. To some extent, these procedures preconditioned the projects to spend more time on nonprogramming activities than the typical small application software project. Although the participants' feedback indicated that these procedures were key to the projects' success, it is important to note that the confirmation of Hypothesis 1 and the nonconfirmation of Hypotheses 2 and 3 are somewhat preconditioned by the projects' procedures. It would be highly valuable to attempt an independent replication of the experimental results, under the condition that one of the teams develop their product without using the procedures established here.

Acknowledgment

I would like to acknowledge the enthusiasm, talent, and dedication of the participants in this experiment: project managers P. Jansma and L. Duclos, and team members E. Babcock, K. Beck, T. Everman, J. Han, N. Law, J. Painter, S. Peng, S. Tan, and B. Wu. I would also like to thank USC Computer Science Professors L. Cooprider and J. Guttag for their thorough acceptance tests of the products, and Prof. E. Horowitz, USC Computer Science Department Head, for his encouragement and support in a number of technical and administrative dimensions.

References

[1] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[2] R. Nelson, *Software Data Collection and Analysis at RADC.* Rome, NY: Rome Air Develop. Center, 1977.
[3] C. P. Felix and C. E. Walston, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, 1977.
[4] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, July 1978.
[5] B. W. Boehm and R. W. Wolverton, "Software cost modeling: Some lessons learned," in *Proc. U.S. Army/IEEE Second Life Cycle Management Workshop*, Aug. 1978.
[6] W. A. Hosier, "Pitfalls and safeguards in real-time digital systems with emphasis on programming," *IRE Trans. Eng. Management*, pp. 99–115, June 1961.
[7] J. R. Brown, "The impact of modern programming practices on software development," RADC-TR-77-121, June 1977.
[8] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, pp. 615–636, June 1974.
[9] F. P. Brooks, *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.
[10] E. F. Bardain, "Research studies of programmers and programming," 1964; cited in D. B. Mayer and A. W. Stalnaker, "Selection and evaluation of computer personnel," in *Proc. ACM68*, 1968, pp. 657–670.
[11] F. S. Ingrassia, "Combating the '90% complete' syndrome," *Datamation*, pp. 171–176, Jan. 1978.
[12] T. E. Bell and T. A. Thayer, "Software requirements: Are they really a problem?," in *Proc. 2nd Int. Conf. Software Eng.*, IEEE Comput. Soc., Oct. 1976, pp. 61–68.
[13] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. Ass. Comput. Mach.*, pp. 760–768, Sept. 1978.
[14] P. Naur, "Programming by action clusters," *BIT*, vol. 9, no. 3, pp. 250–258, 1969.
[15] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, pp. 156–173, June 1975.
[16] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability.* North-Holland, 1978.
[17] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, pp. 1226–1241, Dec. 1976.
[18] K. Freburger and V. R. Basili, "The software engineering laboratory: Relationship equations," Univ. Maryland Tech. Rep. TR-764, May 1979.
[19] *Proceedings, Government/Industry Software Sizing and Costing Workshop*, U.S. Air Force Electron. Syst. Div., Bedford, MA, Oct. 1974.

Barry W. Boehm received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957 and the M.A. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, respectively.

From 1978–1979 he was a Visiting Professor of Computer Science at the University of Southern California. He is currently a Visiting Professor at the University of California, Los Angeles, and Director of Software Research and Technology in TRW's Software Systems Division. He was previously Head of the Information Sciences Department at The Rand Corporation, and Director of the 1971 Air Force CCIP-85 study. His responsibilities at TRW include direction of TRW's internal software R&D program, of contract software technology projects, of the TRW software development policy and standards program, of the TRW Software Cost Methodology Program, and of the TRW Software Productivity Program. He has recently written a book, *Software Engineering Economics*, being published by Prentice-Hall.